9 This HTML page is also a PDF which is also a ZIP which is also a Ruby script which is an HTTP quine; or, The Treachery of Files

by Evan Sultanik
from a concept independently conceived by Ange Albertini
and with great technical assistance from Philippe Teuwen

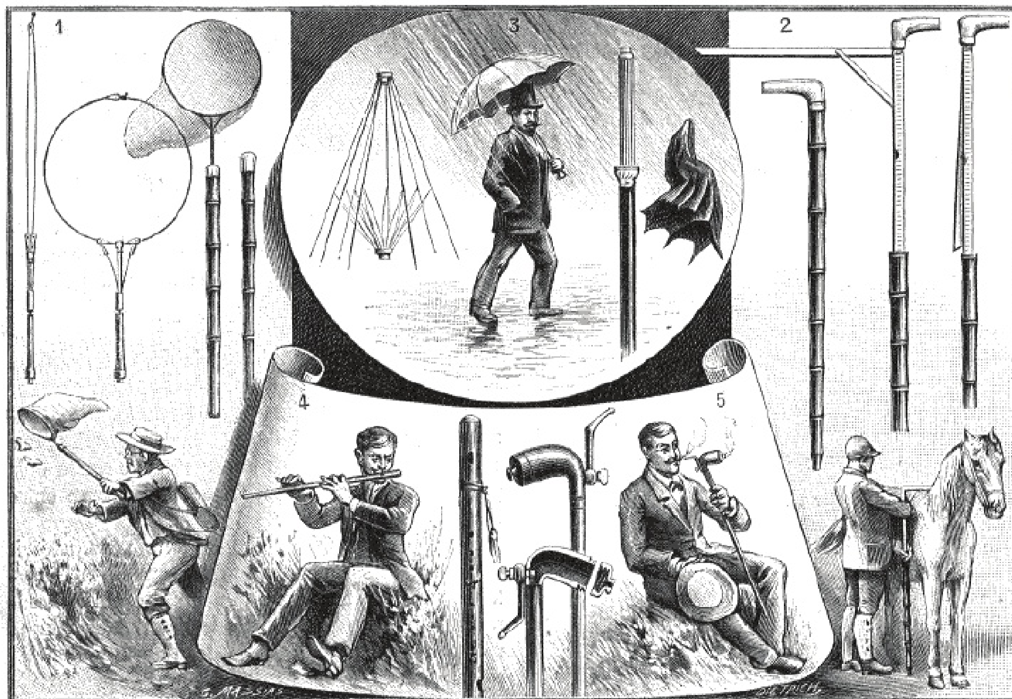
Please rise and open your hymnal for the recitation of PoC||GTFO 7:6.

> "A file has no intrinsic meaning. The meaning of a file—its type, its validity, its contents—can be different for each parser or interpreter."

You may be seated.

In the spirit of самиздат and the license of this publication, we thought it might be nifty to aid its promulgation by enabling the PDF to mirror itself. That's right, this PDF is an HTTP quine: it is a web server that serves copies of itself.

```
$ ruby pocorgtfo11.pdf &
Listening for connections on port 8080.
To listen on a different port,
re-run with the desired port as a command-line argument.
$ curl -s http://localhost:8080/pocorgtfo11.pdf | diff -s - pocorgtfo11.pdf
A neighbor at 127.0.0.1 is requesting /pocorgtfo11.pdf
Files - and pocorgtfo11.pdf are identical
```



Utilisation de la canne. — 1. Canne-filet à papillons. — 2. Canne à toiser les chevaux. — 3. Canne-parapluie. — 4. Canne musicale. — 5. Ceci n'est pas une pipe.

This polyglot once again exploits the fact that PDF readers ignore everything before the first instance of "%PDF". Coupled with Ruby's `__END__` token—which effectively halts interpretation—and its `__FILE__` token—which resolves to the path of the file being interpreted—it's actually quite easy to make an HTTP quine by prepending the PDF with the following:

```ruby
require 'socket'
server = TCPServer.new('', 8080)
loop do
        socket = server.accept
        request = socket.gets
        response = File.open(__FILE__).read
        socket.print "HTTP/1.1 200 OK\r\n" +
                "Content-Type: application/pdf\r\n" +
                "Content-Length: #{response.bytesize}\r\n" +
                "Connection: close\r\n"
        socket.print "\r\n"
        socket.print response
        socket.close
end
__END__
```

But why stop there? Ruby makes all of the bytes in the script that occur after the `__END__` token available in the special "`DATA`" object. Therefore, we can add additional content between `__END__` and `%PDF` that the script can serve.

```ruby
require 'socket'
server = TCPServer.new('', 8080)
html = DATA.read().split(/<\/html>/)[0]+"</html>\n"
loop do
        socket = server.accept
        if socket.gets.split(' ')[1].downcase.end_with? ".pdf" then
                c = "application/pdf"
                d = File.open(__FILE__).read
                n = File.size(__FILE__)
        else
                c = "text/html"
                d = html
                n = html.length
        end
        socket.print "HTTP/1.1 200 OK\r\nContent-Type: #{c}\r\nContent-Length: #{n}\r\nConnection: close\r\n\r\n"+d
        socket.close
end
__END__
<html>
  <head>
    <title>An HTTP Quine PoC</title>
  </head>
  <body>
    <a href="pocorgtfo11.pdf">Download pocorgtfo11.pdf!</a>
  </body>
</html>
```

Any HTTP request with a URL that ends with `.pdf` will result in a copy of the PDF; anything else will result in the HTML index parsed from `DATA`.

Since the data between `__END__` and `%PDF`... is pure HTML already, it would be a shame not to make this file a pure HTML polyglot, too (similar to PoC||GTFO 0x07). Doing so is relatively simple by wrapping PDF in HTML comments:

```
INSERT RUBY WEB SERVER HERE
__END__
<html>
  ...
</html>
<!--
INSERT RAW PDF HERE
-->
```

This is valid Ruby, since Ruby does not interpret anything after the `__END__`. The PDF does not affect the validity of the HTML since it is commented. There will be trouble if the byte sequence "`-->`" (`2D 2D 3E`) occurs anywhere within the PDF, but this is very unlikely and has proven not to be a problem.

Wrapping the Ruby webserver code in an HTML comment would have been ideal, and does in fact work for most PDF viewers. However, the presence of an HTML opening comment before the `%PDF` causes Adobe's parser to classify the file as HTML and therefore refuse to open it.

Unfortunately, some web browsers interpret the Ruby code as having an implied "`<html>`" preceding it, adding all of that text to the DOM. This is remedied with Javascript in the HTML that sanitizes the DOM if necessary.

As has become the norm, this PDF is also a valid ZIP. This feat does not affect the Ruby/HTML portion since the ZIP is embedded later in the file as an object within the PDF (cf. PoC||GTFO 1:5). This presents an additional opportunity for the webserver: if the script can unzip itself, then it can also serve all of the contents of the ZIP. Unfortunately, Ruby does not have a ZIP decompression facility in its standard library. Therefore, the webserver calls the `unzip` utility with the "`-l`" option, parsing the output to determine the names and sizes of the constituent files. Then, a call to `unzip` with "`-p`" writes raw decompressed contents to STDOUT, which the web server splits apart and stores in memory. Any HTTP request with a URL that matches a

file path within the ZIP is served that decompressed file. This allows us to have images like a `favicon` in the HTML. In the event that the PDF is interpreted as raw HTML—*i.e.*, it was *not* served from the Ruby script—a Javascript function conveniently hides all of the ZIP access portions.

With all of this feature bloat, the Ruby/HTML code that is prepended before the PDF started getting quite large. Unfortunately, some PDF readers like PDFium[16] (the default PDF viewer shipped with Chrom(e|ium)) fail unless they find "`%PDF`" within the first 1024 characters. Therefore, the final trick in this polyglot is to exploit Ruby's multiline comment syntax (which, like the `__END__` token, owes itself to Ruby's Perl heritage). This allows us to start the PDF header early, within a comment that will not be interpreted. Within that PDF header we open a dummy object stream that will contain the remainder of the Ruby script and the following HTML code before the start of the "real" PDF.

```
require 'socket'
=begin
%PDF-1.5
9999 0 obj
<<
/Length INSERT_#
    _REMAINING_RUBY_AND_HTML_BYTES_HERE
>>
stream
=end
INSERT REMAINING RUBY CODE HERE
__END__
INSERT HTML HERE
<!--
endstream
endobj
INSERT RAW PDF HERE WITH LEADING %... HEADER
    REMOVED
-->
```

Figure 5 describes the anatomy of the polyglot, as interpreted in each file format.

[16] https://pdfium.googlesource.com/pdfium/

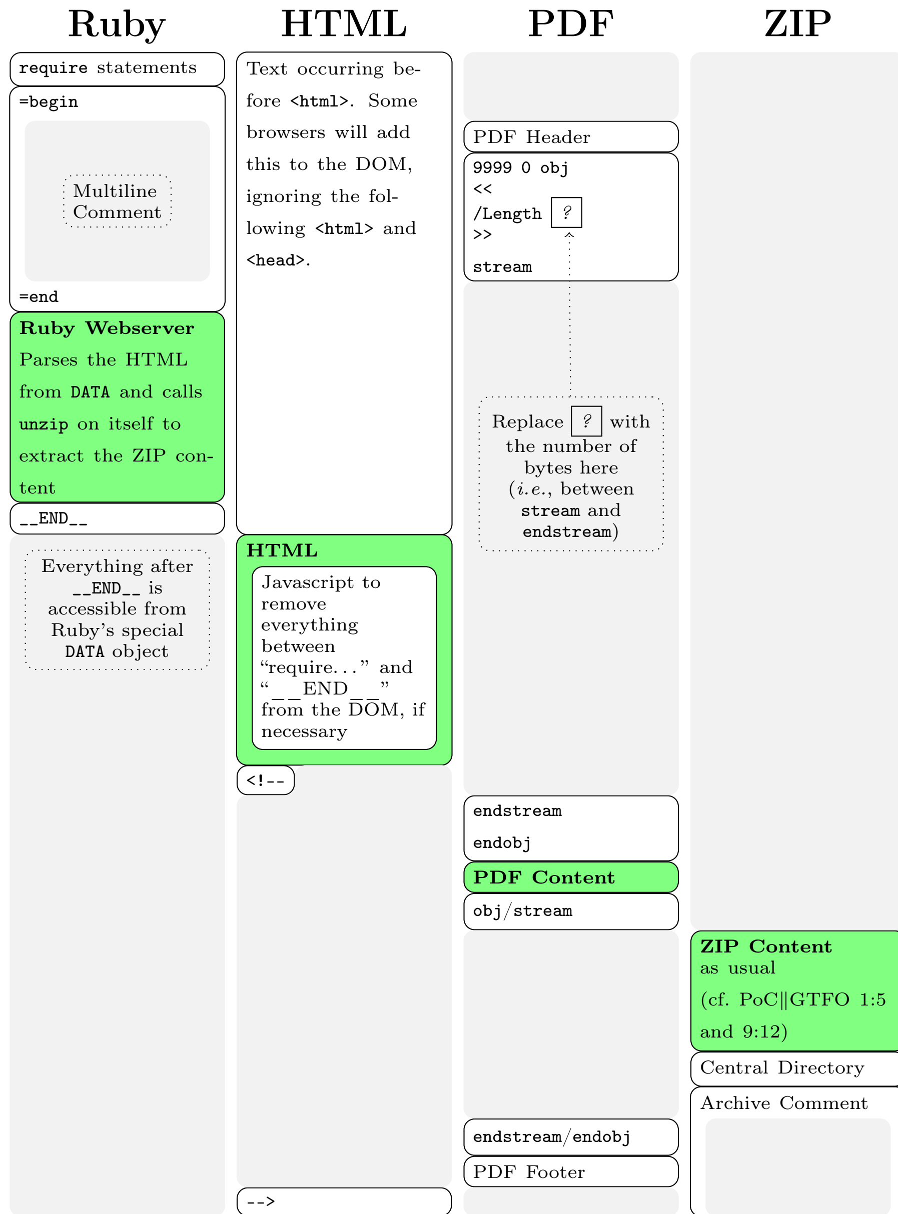


Figure 5 – Anatomy of the Ruby/HTML/PDF/ZIP polyglot. Green portions contain the main content of their respective filetypes. White portions are for context and to illustrate modifications necessary to make the polyglot work. Gray portions are not interpreted by their respective filetypes.